Towards a New Formal SDL Semantics based on Abstract State Machines

U. Glässer[a], R. Gotzhein[b] and A. Prinz[c]

[a]Heinz Nixdorf Institute, Paderborn University, Dept. of Math. & Computing Science, D-33102 Paderborn, Germany, glaesser@uni-paderborn.de

[b]Computer Networks Group, University of Kaiserslautern, Dept. of Computing Science, D-67653 Kaiserslautern, Germany, gotzhein@informatik.uni-kl.de

[c]DResearch GmbH Berlin, D-10319 Berlin, Germany, prinz@dresearch.de

With the year 2000 approaching, a new version of SDL called SDL-2000 is currently reaching maturity, and is expected to pass the standardization bodies shortly. Apart from the usual language maintenance, SDL-2000 will offer new features for exception handling and object-oriented data types. To capture these features formally, a new formal SDL semantics is being devised. In several meetings of ITU-T SG10/Q6, the essential design objectives have been clarified, and an outline of the behaviour model for SDL has been presented and discussed. A major concern in this discussion has been the demand for an executable model, which calls for an operational formalism with readily available tool support. Subsequent investigations have shown that *Abstract State Machines (ASMs)* meet this and all other design objectives, and therefore have been chosen as the underlying formalism. In this paper, ASMs are applied to define the behaviour model of a sample SDL specification formally, thereby illustrating the approach in general.

1. INTRODUCTION

In 1988, a formal semantics for SDL has been added as Annex F to the Z.100 SDL standard of ITU-T [12]. Along with the efforts to improve SDL, this semantics has been revised several times since then. Essentially, Annex F defines a sequence of Meta IV programs that take an SDL specification as input, determine the correctness of its static semantics, perform a number of transformations to replace several SDL language constructs, and interpret the specification. It has been argued that this style of defining the formal semantics is particularly suitable for tool builders.

With the ongoing work to improve SDL, and a new version called SDL-2000 to pass the standardization bodies shortly, it has become apparent that a decision concerning the SDL semantics needs to be taken. Based on the assessment that the existing Meta IV programs will be too difficult to maintain, there is a strong tendency towards defining a new formal semantics. In several meetings of ITU-T SG10/Q6, the essential design objectives of such a semantics have been clarified (see also [7]). Among the primary design objectives is *intelligibility*, which is to be achieved by building on well-known mathematical models and notation, a close correspondence between specification and underlying model, and by a concise and well-structured semantics document. Of similar importance and causally dependent on intelligibility is *maintainability*, since SDL is an evolving language. Language modifications currently under consideration include exception handling, hierarchical states, dynamic block creation, object-oriented data types, real time expressiveness beyond timers, and the removal of some obsolete language features. Therefore, the semantic model has to be sufficiently rich and flexible.

During the discussions within ITU-T SG10/Q6, it has turned out that a prime design objective is the demand for an *executable semantics*. This calls for an operational formalism with readily available tool support. Subsequent investigations have shown that *Abstract State Machines* (ASMs) meet this and all other design objectives, and therefore have been chosen as the underlying formalism. In this paper, we show how ASMs are applied to define the behaviour model of a sample SDL specification formally. More specifically, we develop a behaviour model that can be understood as ASM code generated from an SDL specification. This approach differs substantially from the interpreter view taken in previous work (see also [5,4]), and will enable SDL-to-ASM compilers.

We define a distributed execution model for SDL based on the formalism of *multi-agent real-time ASM*. The behaviour of active SDL objects is described in terms of concurrently operating and asynchronously communicating ASM agents directly reflecting the intuitive understanding of Z.100. Due to its abstract operational view, one obtains a clear and concise formalization of the dynamic semantics of SDL at a natural level of detail and precision.

For the purpose of illustrating the approach, a formal behaviour model of a sample SDL system specification is presented (cf. [7]). Section 2 gives an overview of the modeling approach and illustrates the ASM representation of SDL specifications which serves as a starting point for defining the dynamic semantics in Section 3 (signal flow model) and Section 4 (control flow model). Section 5 contains the conclusions.

2. BACKGROUND AND OVERVIEW

The material of this section is intended to provide an informal overview of the modeling approach, without going into detail. For a rigorous definition of the mathematical foundations, the reader is referred to [8,9]. A comprehensive treatment of methodological aspects of ASM-based modeling and validation of complex systems can be found in [2].[1]

2.1. Associating ASM models with SDL specifications

The dynamic semantics associates with each SDL specification, represented as an abstract syntax tree (AST), a particular multi-agent real-time ASM. Intuitively, an ASM consists of a set of *autonomous agents* cooperatively performing *concurrent machine runs*. The behaviour of agents is determined by *ASM programs*, each consisting of a set of *transition rules*, which define the set of possible runs. Each agent has its own *partial view* on a given global state, on which it fires the rules of its program. According to this view, agents have a *local state* and a *shared state* through which they can interact.

[1] For a comprehensive overview on ASM applications and other available material see also the annotated ASM bibliography [1] as well as the following two URLs: `http://www.uni-paderborn.de/cs/asm/` and `http://www.eecs.umich.edu/gasm/`.

It is pointed out that there are strong structural similarities between SDL systems and ASMs, which is a prerequisite for intelligibility and maintainability. These structural similarities are now exploited by identifying ASM agents with certain SDL units. For instance, agents will be introduced for processes, process sets, blocks, and channel segments. Agents can be created during the system initialization phase as well as dynamically, which allows, e.g., to directly represent dynamic process creation in the underlying model.

The execution of a system starts with the creation of a single agent for the SDL unit "system". This agent then creates further agents according to the substructure of the system as defined by the specification, and associates a view and an ASM program with each of them. ASM programs are determined by the kind of the SDL unit modeled by a given agent and the AST of a given SDL specification.

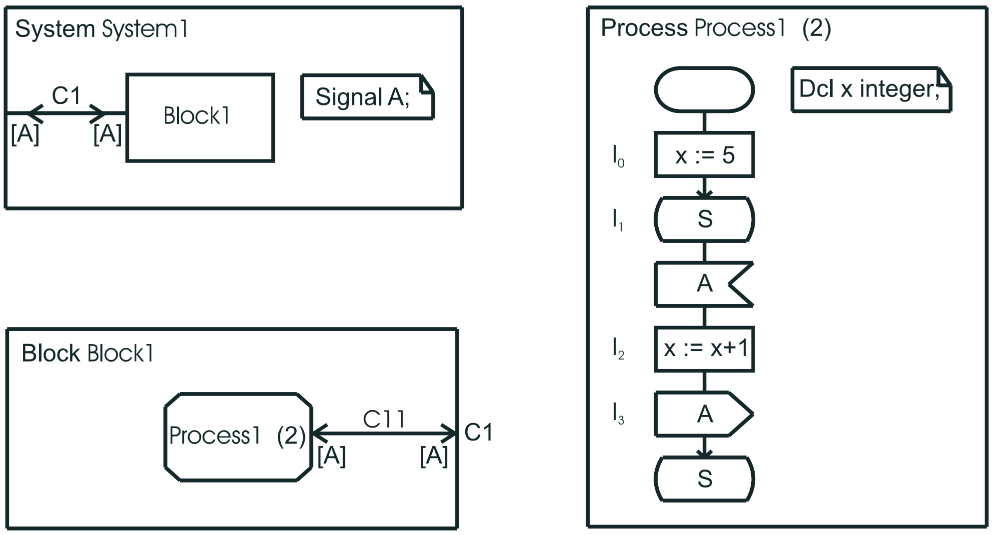

Figure 1. An SDL system specification

Example In Figure 1, a system consisting of a single block Block1 and a delaying channel segment C1 is specified in SDL. The dynamic semantics defines that execution of this system starts with the creation of an agent for System1 with an associated ASM program. This program consists of a rule that, when fired, leads to the creation of two further agents, one agent for Block1 and another for C1. After this rule has been fired and the system has reached an initial state, the agent modeling System1 becomes inactive.

Agents modeling SDL units with a substructure such as block, process set, or process, create further agents modeling the SDL units of this substructure. The ASM program associated with these agents determines the set of agents to be created and their behaviour. Starting with the agent modeling the SDL unit "system", this is repeatedly done until leaf units of the SDL specification such as process or service are reached.

Example In Figure 1, Block1 is substructured into a process set and a channel segment C11. These SDL units are modeled by two corresponding ASM agents. The execution of the ASM agent modeling the process set then leads to the creation of two additional agents, each modeling an instance of Process1. All created processes fire transition rules to reach a specified initial state. This completes the system initialization phase.

Once the initial system structure has been built up by creating a set of agents and associating, with each agent, a view and an ASM program, the actual system execution starts. The behaviour of agents during system execution is determined by their ASM programs defining the set of possible runs. During a run, further agents may be created, and existing agents be terminated, according to the transitions of the SDL processes.

Example The actual execution of the system specified in Figure 1 starts when the agents modeling System1, Block1, C1, C11, and the process set as well as the instances of Process1 have been created and reached their initial state. The behaviour of both process instances is formally defined by an ASM program that is derived from the process graph. It is one of the tasks of the formal semantics definition to make this derivation precise.

2.2. Real-Time Behaviour

By introducing a notion of global system time and imposing additional constraints on machine runs, one obtains a restricted class of ASM, called multi-agent real-time ASM, with agents performing *instantaneous* actions in *continuous* time. For the purpose considered here, it is important that agents fire their rules at the moment they are enabled.

SDL uses the expression **now** to represent the global system time, where the possible values of **now** are given by the predefined SDL sort *Time*[2]. The machine model is therefore extended by a nullary dynamic function *now* taking values in a corresponding domain *TIME*.

$$now: \quad TIME, \qquad TIME \subseteq \mathbb{R}$$

It is intended that the value of *now* changes dynamically without being affected by the machine model. Intuitively, *now* represents physical time.

2.3. The External World

Following an *open system view*, interactions between a system and the external world, e.g. the environment into which the system is embedded, are modeled in terms of various interface mechanisms. Regarding the reactive nature of distributed systems, it is important to clearly identify and precisely state

- assumptions and constraints on the expected behaviour of the external world, and
- how external conditions and events affect the transition behaviour of an ASM model.

This is achieved through a classification of *dynamic* ASM functions into three categories, namely *controlled*, *monitored* and *shared* functions. By definition, controlled functions can only be altered by an ASM agent, whereas monitored functions exclusively belong to the external world and are *read-only* functions within an ASM program. A monitored function may nevertheless change its values from state to state (e.g., the function *now*).

[2] *Time* values are actually real numbers with restricted operations (see Appendix D to Z.100); in particular, SDL does not define a notion for scaling time (it is merely assumed that the same scale of time is used throughout an entire system description).

Shared functions may be altered by the environment as well as by the machine model. Hence, a reasonable *integrity constraint* on shared functions is that no interference with respect to mutually updated locations must occur. Assuming that the external world itself acts like an ASM agent (or a collection of ASM agents), this is ensured by the underlying model of concurrency[3].

2.4. ASM Representation of SDL Specifications

SDL specifications are represented in the ASM model based on their AST. In Figure 2, an excerpt of the AST for the running example is shown. This AST is then encoded by a finite collection of ASM objects from a static domain *DEFINITION* in combination with various functions defined thereon. By means of these functions, the definitions are related to each other and the relevant information can be extracted. Emphasizing the tight relation between SDL specifications and their ASM representation, the terminology and the notation of the AS are reused here. That is, function names denoting AST components in the machine model are identical to the non-terminals used in the AS definition:

System-definition	:	*DEFINITION*
System-name	:	*DEFINITION* $\rightarrow$ *NAME*
Block-definition-set	:	*DEFINITION* $\rightarrow$ *DEFINITION*-**set**
Block-definition	:	*DEFINITION* $\rightarrow$ *DEFINITION*
... *etc.*		

2.5. Composition of the Dynamic Semantics

Following an abstract operational view, behaviour is expressed in terms of *SDL abstract machine runs*. The formal definition of the dynamic semantics is restricted to specifications that comply with the static semantics of SDL. Previous work on an ASM semantics for SDL (see [5,4]) provides a conceptual framework which is partly reused here.

Starting with the native ASM formalism, the dynamic semantics of SDL is defined in two steps. Firstly, an SDL oriented ASM formalism called BSDL (former work on BSDL is reported in [11]) is given (see Section 3). BSDL can be seen as a kernel containing certain fundamental SDL mechanisms, such as signals, gates, channels, signal consumption, and timers. These mechanisms are collectively referred to as "signal flow model", and are independent of a particular SDL specification.

Secondly, the semantics of SDL is defined based on the signal flow model (see Section 4). Starting point is an ASM representation of SDL specifications as outlined in Section 2.4. ASM modules that determine the behaviour of different kinds of SDL components are provided and used to define the execution model of the SDL specification shown in Figure 1. This is collectively referred to as "control flow model".

2.6. Notational Conventions

For an improved readability, complex transition rules are defined in a stepwise manner by means of *macros* (with parameters). Formally, macros are syntactic abbreviations, i.e. each occurrence of a macro in a rule is to be replaced textually by the related macro definition (replacing formal parameters by actual parameters). For further details, see [6].

[3]For details, see the notion of *partially ordered run* in [8].

- *System-definition*
  - *System-name*
    - System1
  - *Block-definition*-**set**
    - *Block-definition*
    - (include AST of block Block1 here)
  - *Process-definition*-**set**
  - *Channel-definition*-**set**
    - *Channel-definition*
      - *Channel-name*
        - C1
      - *Channel-path*
        - *Originating-entity*
          - Block1
        - *Destination-entity*
          - ENVIRONMENT
        - *Signal-identifier*-**set**
          - *Signal-identifier*
            - A
      - *Channel-path*
        - *Originating-entity*
          - ENVIRONMENT
        - *Destination-entity*
          - Block1
        - *Signal-identifier*-**set**
          - *Signal-identifier*
            - A
  - *Signal-definition*-**set**
    - *Signal-definition*
      - *Signal-name*
        - A
  - *Data-type-definition*
  - *Syntype-definition*-**set**

Figure 2. Abstract Syntax Tree of the SDL Specification of System1

3. SIGNAL FLOW MODEL

This section introduces *Base SDL (BSDL)* as a kind of kernel language for dealing with certain fundamental features of SDL at a more abstract level. The main focus here is on signal flow aspects, in particular, on the definition of a basic process pattern stating how *active objects* communicate through signals via *gates*. Also, certain structural aspects are covered here.

3.1. Units

To deal with the various types of behavioural SDL objects (systems, blocks, processes) in a uniform way, the concept of *behavioural units* (*units* for short) is introduced. The intention is that behavioural units look the same from their outside (interface) regardless whether they represent direct behaviour, e.g. as processes, or structural aspects, e.g. as systems. Units communicate asynchronously by means of signals via gates, where the gates are interfaces (see Section 3.3).

There is a domain *PLAINUNIT* of plain units with direct behaviour representation. Furthermore, there is a domain *STRUCTURALUNIT* of structural units, which are composed of plain units or structural units.

$$UNIT \equiv PLAINUNIT \cup STRUCTURALUNIT$$

$$subunits \;:\; STRUCTURALUNIT \rightarrow UNIT\textbf{-set}$$

Plain units usually represent behaviour in the sense of *AGENT*s, whereas structural units usually do not have a behaviour of their own. Instead, their behaviour is defined by the behaviour of their internal units. Internal units communicate with each other and with the outside via their gates. The enclosing structural unit unifies the gates of its internal units and its own gates, thus enabling communication.

3.2. Signals

A static domain *SIGNAL* denotes the set of *signal types* occurring in an SDL specification. Dynamically created *signal instances* (*signals* for short) are given by a dynamic domain *SIGNALINST*, where each element of *SIGNALINST* is uniquely related to an element of *SIGNAL*. Finite *sequences* of signals belong to a domain $SIGNALINST^*$ with a distinguished element *empty* denoting the empty sequence of signals.

To each signal a (possibly empty) list of *signal values* is assigned. Since the type information and concrete value is immaterial to the dynamic aspects considered here, signal values are abstractly represented in a uniform way as elements of some static domain *VALUE*. We introduce values at this point unstructured and with no type information.

$$name: \; SIGNALINST \rightarrow SIGNAL\,, \quad values: \; SIGNALINST \rightarrow VALUE^*$$

Additional functions on signals are *sender*, *toArg*, and *viaArg* specifying the sender process, the destination and optional constraints on admissable communication paths. The meaning of these functions is detailed in subsequent sections.

3.3. Gates

Exchange of signals between scope units (such as processes, blocks or a system and its environment) is modeled by means of *gates*. A gate forms an interface for *serial* and *unidirectional* communication between two or more units. The same gate may have the role of an *input gate* for one unit and the role of an *output gate* for another unit. To every unit a set of input gates and a set of output gates is associated.

$$ingates, outgates: \ UNIT \rightarrow GATE\text{-}\mathbf{set}\,, \quad \forall a \in UNIT: \ ingates(a) \cap outgates(a) = \emptyset$$

3.3.1. Discrete Delay Model

A gate must be capable of holding signals that arrived there. Hence, to each gate a (possibly empty) *signal queue* is assigned, as detailed below. Moreover, signals need not reach their destination instantaneously but may be subject to delays. That means, it must be possible to send signals to arrive in the future. Although those signals are only available for disposal when their arrival time has come, they are represented within gates.

To model this behaviour, each signal S which is in transit (not yet arrived) has its individual *arrival time*, i.e. the time at which S eventually reaches a certain gate.

$$arrival \ : \ SIGNALINST \rightarrow TIME$$

One can now represent signals which are in transit to a gate g by means of a *schedule* specifying the respective *signal arrivals* at the gate. More generally, in a given state of the abstract machine model, one can associate to every gate g in $GATE$ some finite sequence of signals, called $schedule(g)$, such that $schedule(g)$ specifies all those signals S which are currently in transit (with destination g).

A reasonable integrity constraint on $schedule(g)$ is that signals in $schedule(g)$ are linearly ordered by their arrival times. That is, if $schedule(g)$ contains signals S, S' and $arrival(S) < arrival(S')$ then $S < S'$ in the order defined by $schedule(g)$.

Intuitively, a signal S in $schedule(g)$ does not arrive at gate g before $now \geq arrival(S)$. In other words, S remains 'invisible' as long as it is in transit. Thus, in every given state of the abstract machine model, the visible part of $schedule(g)$ forms a possibly empty signal queue, denoted as $queue(g)$, which is defined as follows (see also Figure 3).

$$schedule, \ queue \ : \ GATE \rightarrow SIGNALINST^*$$

$$queue(g) \ \equiv \ \langle s : s \ in \ schedule(g) \wedge now \geq arrival(s) \rangle$$

3.3.2. Operations on Schedules

To ensure that the order on signals is preserved when new signals are added to the schedule of a gate, there is a special insertion function on schedules.

$$insert \ : \ SIGNALINST \times TIME \times SIGNALINST^* \rightarrow SIGNALINST^*$$

The result of inserting some signal S with arrival time t into a schedule seq is defined as follows.

$$insert(S, t, seq) =$$

$$\begin{cases} \langle S | seq \rangle, & \text{if } \ seq = empty \ \text{ or } \ t < arrival(seq.head), \\ \langle seq.head | insert(S, t, seq.tail) \rangle, & \text{otherwise.} \end{cases}$$

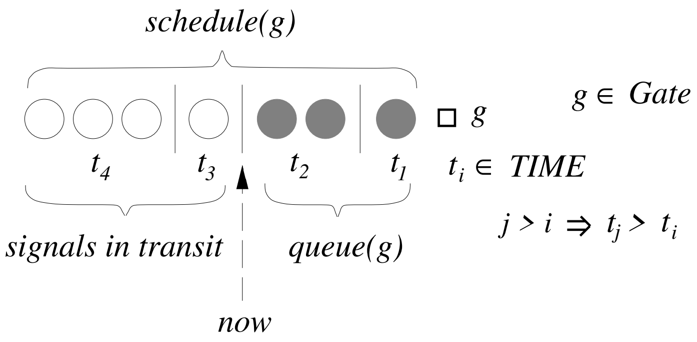

Figure 3. Signals at a gate

Analogously, a function *delete* (with an obvious meaning) is used to remove an item from a schedule.

$$delete \quad : \quad SIGNALINST \times SIGINST^* \rightarrow SIGINST^*$$

As shorthands for operations on schedules to be used in subsequent rules the following two macros are defined.

$$\begin{aligned} \text{INSERT}(S, t, g) &\equiv schedule(g) := insert(S, t, schedule(g)), \quad arrival(S) := t \\ \text{DELETE}(S, g) &\equiv schedule(g) := delete(S, schedule(g)), \quad arrival(S) := undef \end{aligned}$$

3.4. Channels

Channels are decomposed into individual channel paths having the role of connection primitives with associated gates. More specifically, a channel path forms a unidirectional point-to-point communication link for the transport of signals from its ingate ($from$) to its outgate (*to*). Channel paths declared in an SDL specification are formally represented by the elements of a static domain *LINK*.

$$LINK \subseteq AGENT$$

SDL considers channels as reliable and order-preserving connections. A channel may however delay the transportation of a signal for an *indeterminate* and *non-constant* time interval. Although the exact delaying behaviour is not further specified, the fact that channels are reliable implies that all delays must be finite.

Signal delays are modeled through a monitored function *delay* stating the dependency on external conditions and events. That is, *delay* dynamically associates time intervals from a domain *DURATION* to the elements of *LINK*, where the length of a time interval is determined by the environment. Taking into account that there are also non-delaying channels, a reasonable integrity constraint on the function *delay* is that the result value on non-delaying channel paths is always 0.

Channel paths are further characterized by a set of signal types they are able to convey. This constraint is stated by a function *with* defined on *LINK*.

$$\begin{array}{lll} \mathit{from}, \mathit{to} & : & \mathit{LINK} \rightarrow \mathit{GATE} \\ \mathit{with} & : & \mathit{LINK} \rightarrow \mathit{SIGNAL}\textbf{-set} \\ \mathit{delay} & : & \mathit{LINK} \rightarrow \mathit{DURATION} \end{array}$$

Channel Behaviour

A link agent l performs a single operation: signals received at $\mathit{from}(l)$ are forwarded to $\mathit{to}(l)$. That means, l continuously watches ingate $\mathit{from}(l)$ waiting for the next deliverable signal in $\mathit{queue}(\mathit{from}(l))$. Whenever l is applicable to a signal S identified by the head of $\mathit{queue}(\mathit{from}(l))$, it attempts to discard S from $\mathit{queue}(\mathit{from}(l))$ in order to insert S into $\mathit{schedule}(\mathit{to}(c))$. Note that this attempt need not necessarily be successful as, in general, there may be several link agents competing for the same signal S.

How does a link agent l know whether it is applicable to a signal S? Now, this decision depends on the values of $\mathit{toArg}(S)$, $\mathit{viaArg}(S)$, $\mathit{name}(S)$ and $\mathit{with}(l)$. In other words, l is a legal choice for the transportation of S only, if $\mathit{name}(S) \in \mathit{with}(l)$ and there is an admissible path connecting $\mathit{to}(l)$ to a valid destination of S. Abstractly, this decision can be expressed using a predicate *Compatible* as stated below.[4]

The behaviour of channel paths is specified by the module Link_Module below, where an auxiliary function *last_time* defined on *LINK* states the accumulated delays as required to ensure that the transfer of signals is *order-preserving*.

Link_Module:

```
FORWARDSIGNAL
  ≡ if Self.from.queue ≠ empty then
       let S = Self.from.queue.head in
          if Applicable(Self, S) then
             DELETE(S, Self.from)
             INSERT(S, Delay(Self), Self.to)
             Self.last_time := Delay(Self)
    where
       Applicable(Self, S)
          ≡ S.name ∈ Self.with ∧ Compatible(Self.to, S.name, S.toArg, S.viaArg)
       Delay(Self)
          ≡ max(now + Self.delay, Self.last_time)
```

3.5. Processes

A static domain *PROCESS* represents the *process definitions* as introduced by an SDL specification, where properties associated with elements of this domain are stated by various functions defined on *PROCESS*.

[4] Clearly, there is quite some amount of complexity hidden in this predicate *Compatible* performing the computations as defined with the reachability rules of [12]. However, all involved functions are computable and could (and probably will) be refined down to the level of detail and precision stated in [12].

Individual process instances are represented by a dynamic domain *PROCESSINST*, $PROCESSINST \subset AGENT$, which grows and shrinks dynamically as new process instances are created at run time respectively old ones cease to exist. Each process instance has a unique *process instance identifier – PId* from a dynamic domain *PID* as stated by a corresponding function *self* defined on process instances. Input ports of process instances are modeled as finite sequences of signals in terms of a gate.

$$\mathit{self} \ : \ \mathit{PROCESSINST} \rightarrow \mathit{PID}, \quad \mathit{inport} \ : \ \mathit{PROCESSINST} \rightarrow \mathit{GATE}$$

Process instances are organized in *process instance sets*, where all members of a process instance set share the same sets of input gates and output gates. A process instance set is abstractly represented by an element of a dynamic domain *PROCESSSET*. The relation between process instances and process sets is modeled by a dynamic function *owner*. Signals received at an input gate of a process set are appended to the input port of a process instance depending on the value of the *toArg*. Simultaneously arriving signals which match the same process instance are appended, one at a time, in an order chosen non-deterministically. Signals are discarded whenever no matching receiver instance exists. Notice that SDL allows the following syntactic ambiguity (where the value of *toArg* may also be undefined).

$$\mathit{toArg} \ : \ \mathit{SIGNALINST} \rightarrow \mathit{PID} \cup \mathit{PROCESS}$$

Process agents frequently check their input gates for new signals as specified in terms of a rule macro HandleSignals.

$$
\begin{array}{l}
\textsc{HandleSignals} \\
\quad \equiv \texttt{choose } s : \ \mathit{Arrived}(S) \wedge \mathit{Matches}(S) \\
\qquad\quad \textsc{Insert}(\mathit{Self.buffer}, S.\mathit{arrival}, S) \\
\qquad // \text{ discard zombie signals} \\
\qquad \texttt{choose } S : \ \mathit{Arrived}(S) \wedge \mathit{PostMortem}(S) \\
\qquad\quad \textsc{Delete}(\mathit{Self.buffer}, S) \\
\qquad \texttt{where} \\
\qquad\quad \mathit{Arrived}(S) \equiv \ \exists g : g \in \mathit{Self.ingates} \wedge S = g.\mathit{queue.head} \\
\qquad\quad \mathit{Matches}(S) \equiv \ S.\mathit{toArg} = \mathit{Self.pid} \vee S.\mathit{toArg} = \mathit{Self.owner} \\
\qquad\quad \mathit{PostMortem}(S) \\
\qquad\qquad \equiv \ S.\mathit{toArg} \in \mathit{PID} \wedge (\forall p \in \mathit{PROCESSINST} : \mathit{pid}(p) \neq S.\mathit{toArg})
\end{array}
$$

3.5.1. Signal Consumption

Consider a process agent denoted by *Self* consuming input signals from its input port *Self.inport.queue*. The input operation is modeled below, where the parameters *InputSet* and *SaveSet* specify the set of valid input signals and the set of signals to be saved depending on the current state node. Note that an *input node* is defined by a triple (*Signal*, *Vars*, *Trans*) identifying a *signal type*, a (possibly empty) list of *variable names* and a resulting *transition*. The value of *Trans* specifies the label of the transition rule to be executed in the next machine step, as explained in Section 4.

An auxiliary function *extract* (defined below) checks the input port for a suitable input signal S returning *undef*, if no such signal exists. A monitored predicate *Spontaneous* indicates a spontaneous transition event.

$\textsc{ConsumeInputSignal}(\mathit{InputSet}, \mathit{SaveSet})$
$\equiv$ `if` *Spontaneous* `then`
  `choose` $node$: $node \in \mathit{InputSet} \wedge node = \langle none, empty, \mathit{Trans}\rangle$
    $label(\mathit{Self}) := \mathit{Trans}$
 `else`
  `if` $\mathit{Self.inport.queue} \neq \mathit{emptyQ}$ `then`
   `let` $S = extract(\mathit{Self.inport.queue}, \mathit{SaveSet})$ `in`
    `if` $S \neq undef$ `then`
     $\textsc{Delete}(S, \mathit{Self.inport})$
     `let` $\langle S.name, \mathit{Vars}, \mathit{Trans}\rangle \in \mathit{InputSet}$ `in`
      $label(\mathit{Self}) := \mathit{Trans}$
      $\textsc{AssignValues}(S, \mathit{Vars}, \mathit{Self})$

The auxiliary function *extract* is defined recursively as follows.

$$extract(Q, \mathit{SaveSet}) =$$
$$\begin{cases} undef, & \text{if } Q = empty, \\ S, & \text{if } Q = \langle S|\mathit{RestQ}\rangle \wedge name(S) \notin \mathit{SaveSet}, \\ extract(\mathit{RestQ}, \mathit{SaveSet}), & \text{if } Q = \langle S|\mathit{RestQ}\rangle \wedge name(S) \in \mathit{SaveSet}, \end{cases}$$

3.5.2. Timers

A particular concise way of modeling timers is by identifying timer objects with respective timer signals. More precisely, each *active* timer is represented by a corresponding timer signal in the schedule associated with the input port of the related process instance.

$$\mathit{TIMER} \subseteq \mathit{SIGNAL}, \quad \mathit{TIMERINST} \subseteq \mathit{SIGNALINST}$$

Timer signals of a given process are uniquely identified by their timer name in combination with a list of timer values as stated by the following function.

$$tsig \;:\; \mathit{PID} \times \mathit{TIMER} \times \mathit{VALUE}^* \rightarrow \mathit{TIMERINST}$$

The information associated with timers is accessed using the functions defined on *SIGNAL* with their usual meaning (where both *toArg* and *viaArg* have the value *undef*).

To indicate whether a timer instance is active or passive, there is a corresponding predicate *Active*. For a given process *PId* and a given timer t, the value of *Active* is defined as follows.

$$Active(PId, t) \equiv\; t.arrival > now \vee t \;\mathit{in}\; PId.inport.queue$$

The macros below model the operations *set* and *reset* on timers as executed by a process agent of the abstract machine model. An SDL **set** will be transformed into a reset operation immediately followed by a set operation.

```
SetTimer(TName, TValues, Time)
  ≡ let S = tsig(Self, TName, TValues) in
       if Time = undef then
          // set default duration
          Insert(S, defaultTime, Self.inport)
       else
          Insert(S, Time, Self.inport)
    where
       defaultTime ≡ now + duration(TName)

ResetTimer(TName, TValues)
  ≡ let S = tsig(Self, TName, TValues) in
       if Active(Self, S) then
          Delete(S, Self.inport)
```

4. CONTROL FLOW MODEL

Building on the signal flow concepts of BSDL, defined in Section 3, this section complements the semantic definition by fixing the missing control flow aspects.

4.1. SDL Abstract Machine Model

Behavioural properties of ASM agents are expressed in terms of transition rules defined by a finite collection of program *modules*. A module is identified by its module name from a static domain *MODULE*. A dynamic function *Mod* specifies for each agent the module to be executed in the next step of the SDL abstract machine.

$$Mod \;:\; AGENT \rightarrow MODULE$$

An SDL abstract machine run splits into two basically different phases, an *initialization phase* followed by an *execution phase*. Accordingly, there are two kinds of transition rules, namely *initialization rules* and *execution rules*, as identified by two categories of modules.

$$MODULE \;\equiv\; INITMODULE \cup EXECMODULE$$

Initialization modules are denoted by a fixed set of module names as given by the following static domain.

$$INITMODULE \equiv \{\,\mathsf{Init\_Block\_Module},\ \mathsf{Init\_Process\_Set\_Module},\ \mathsf{Init\_Process\_Module}\,\}$$

Execution modules are denoted by the elements of a static domain *EXECMODULE*. Note that the particular set of execution modules depends on the underlying SDL specification. More specifically, there is one execution module for each type of process. This relation is stated by a static function *execmod*.

$$execmod \;:\; PROCESS \rightarrow EXECMODULE$$

The compilation of SDL processes into ASM transition rules handles control flow information through a *labeling* of process graph nodes. Resulting node labels relate individual process operations to transition rules in the machine model. The effect of state transitions of processes is modeled by firing the related transition rules in an analogous order.

Labels are abstractly represented by a static domain *LABEL*. A dynamic function *label*, defined on process agents, is used to model dynamic rule selection. That is, *label* identifies for each process agent the rule to be fired in a given machine state.

$$label \;:\; PROCESSINST \rightarrow LABEL$$

4.2. Preinitial System State

This section states requirements on the initial state S_0 of the abstract SDL machine model. Formally, S_0 is defined by a set of first-order formulas $\varphi_1, \ldots, \varphi_n$ such that $S_0 \models \varphi_1 \wedge \ldots \wedge \varphi_n$. For brevity, it is assumed that dynamic functions (predicates) yield the value *undef* (*false*) on those locations which have not explicitly been initialized with some other value. With regard to dynamic domains that means that their default interpretation is the empty set.[5]

STARTLABEL denotes a set of distinguished *start labels*, one for each process type. The assignment of start labels to process types is stated by a static function *startlabel*.

$$startlabel \;:\; PROCESS \rightarrow STARTLABEL\,, \quad STARTLABEL \subset LABEL$$

Initial Agents

Agents have two basic operation modes as stated by a static domain *MODE*. A unary dynamic function *mode* defined on agents indicates the mode of an agent in a given abstract machine state.

$$MODE \equiv \{\, initial,\ starting \,\}\,, \quad mode \;:\; AGENT \rightarrow MODE$$

Initially, there is a single agent *system* denoting a uniquely determined system instance from $SYSTEMINST = \{\, system \,\}$.

$$Mod(system) = \mathsf{Init\_Block\_Module}\,, \quad mode(system) = initial$$

A unary function *ref* defined on agents identifies for each agent an AST (sub-) definition to be processed during the initialization phase.

$$ref \;:\; AGENT \rightarrow DEFINITION, \quad ref(system) = System\text{-}definition$$

Gates at system level are to be created during the initialization phase. The two sets of system gates are initialized accordingly.

$$ingates(system) = outgates(system) = \emptyset$$

[5] Technically speaking, ASM states form many-sorted structures on top of elementary structures modeling domains through unary predicates.

4.3. System Initialization Phase

Starting from S_0 the initialization rules describe a recursive *unfolding* of the specified system instance according to its hierarchical structure. Each initialization step may create several object instances simultaneously.

An auxiliary function *owner*, defined on units, is introduced in order to state the relation between individual system parts and their constituent components. More specifically, an agent a is considered as owner of another agent b if the component modeled by b actually is a subcomponent of the component modeled by a.

$$owner \;:\; UNIT \rightarrow UNIT$$

4.3.1. System and Block Level Initialization

The initialization of systems and blocks is defined by the module Init_Block_Module, where *Self*, depending on the context, either denotes a system agent or block agent.

Init_Block_Module:

```
InitBlock
  ≡ if mode(Self) = initial then
        mode(Self) := starting
        CreateBlocks(Self, Self.ref.Block-definition-set)
        CreateProcessSets(Self, Self.ref.Process-definition-set)
        CreateConnections(Self, Self.ref.Channel-to-channel-connection-set)
    else
        CreateChannels(Self, Self.ref.Channel-definition-set)
        // switch to execution phase
        Mod(Self) := undef
```

Block Creation

A block instance is represented by an agent of type *BLOCKINST*. The initialization behaviour of block instances is specified by the module Init_Block_Module.

```
CreateBlocks(Owner, Items)
  ≡ do forall item : item ∈ Items
        // create block agent
        CreateBlockAgent(Owner, item)

CreateBlockAgent(Owner, Ref)
  ≡ extend BLOCKINST with b
        ref(b) := Ref
        mode(b) := initial
        owner(b) := Owner
        ingates(b) := ∅
        outgates(b) := ∅
        Mod(b) := Init_Block_Module
```

Process Set Creation

A process set is represented by an agent of type *PROCESSSET*. The initialization behaviour of process sets is specified by the module Init_Process_Set_Module.

$\textsc{CreateProcessSets}(Owner, Items)$
  $\equiv$ `do forall` *item* : $item \in Items$
    // create process agent
    $\textsc{CreateProcSetAgent}(Owner, item)$

$\textsc{CreateProcSetAgent}(Owner, Ref)$
  $\equiv$ `extend` *PROCESSSET* `with` *ps*
    $ref(ps) := Ref$
    $mode(ps) := initial$
    $owner(ps) := Owner$
    $ingates(ps) := \emptyset$
    $outgates(ps) := \emptyset$
    $Mod(ps) :=$ Init_Process_Set_Module

Connection Creation

For each connection declared in a *Channel-to-channel-connection-set* associated with a certain scope unit, depending on whether this connection is unidirectional or bidirectional, either one or two gates of the scope unit need to be created.

An auxiliary function *directions* when applied to a connection determines the specified direction(s).

$directions \ : \ DEFINITION \rightarrow DIRECTION\text{-}\mathbf{set}$

$\textsc{CreateConnections}(Owner, Items)$
  $\equiv$ `do forall` *item* : $item \in Items$
    // create one gate per direction
    `do forall` *direction* : $direction \in directions(item)$
      $\textsc{CreateGate}(Owner, direction)$

$\textsc{CreateGate}(Owner, Direction)$
  $\equiv$ `extend` *GATE* `with` *g*
    $owner(g) := Owner$
    `if` $Direction = in$ `then`
      $ingates(Owner) := ingates(Owner) \cup \{ g \}$
    `if` $Direction = out$ `then`
      $outgates(Owner) := outgates(Owner) \cup \{ g \}$

Channel Creation

Channels are modeled through unidirectional channel paths. Each channel path is represented by an agent of type *LINK*.

$\textsc{CreateChannels}(Owner, Items)$
  $\equiv$ `do forall` *item* : $item \in Items$
    `do forall` *path* : $path \in Channel\text{-}paths(item)$
      $\textsc{CreateChannelPath}(Owner, item, path)$

4.3.2. Process Set Level Initialization

The initialization of process sets is defined by the module Init_Process_Set_Module, where *Self* denotes any process set agent.

Init_Process_Set_Module:

$$
\begin{array}{l}
\textsc{InitProcessSet} \\
\quad \equiv \text{// create process set gates} \\
\qquad \textsc{CreateProcessSetGates}(\mathit{Self}, \mathit{Self.owner.ref}) \\
\qquad \text{// create initial process instances} \\
\qquad \texttt{do forall } i : \ i \in \{ 1 \,..\, \mathit{Self.ref.Initial\text{-}number\text{-}of\text{-}instances} \} \\
\qquad\quad \textsc{CreateProcessInstance}(\mathit{Self}, \mathit{Self.ref}) \\
\qquad \mathit{Mod}(\mathit{Self}) := \mathit{undef}
\end{array}
$$

$$
\begin{array}{l}
\textsc{CreateProcessInstance}(\mathit{Owner}, \mathit{Ref}) \\
\quad \equiv \texttt{extend } \mathit{PROCESSINST} \texttt{ with } p \\
\qquad\quad \mathit{ref}(p) := \mathit{Ref} \\
\qquad\quad \mathit{mode}(p) := \mathit{initial} \\
\qquad\quad \mathit{owner}(p) := \mathit{Owner} \\
\qquad\quad \mathit{Mod}(p) := \mathsf{Init\_Process\_Module}
\end{array}
$$

4.3.3. Process Level Initialization

The initialization of process instances is defined by the module Init_Process_Module, where *Self* denotes any process agent.

Init_Process_Module:

$$
\begin{array}{l}
\textsc{InitProcess} \\
\quad \equiv \mathit{name}(\mathit{Self}) := \mathit{Self.ref.Process\text{-}name} \\
\qquad \mathit{label}(\mathit{Self}) := \mathit{Self.owner.startlabel} \\
\qquad \textsc{CreateProcessGates}(\mathit{Self}) \\
\qquad \textsc{CreateProcessVariables}(\mathit{Self}, \mathit{Self.ref}) \\
\qquad \text{// switch to execution module} \\
\qquad \mathit{Mod}(\mathit{Self}) := \mathit{execmod}(\mathit{Self.owner})
\end{array}
$$

4.4. System Execution Phase

This section extends the description of the behaviour of process agents from the previous Section by introducing another rule, called ExecProcess, as given by the corresponding execution module for a process type. This additional rule defines the behaviour of process agents during the execution phase. Notice that these are the only agents running during the execution phase.

Example Specification

The value of *execmod*(*Process1*) is defined to be Process1_Module. By setting the value of *Mod* to *execmod*(*Process1*), the final step of the initialization of processes, as stated by the InitProcess rule, effectively switches the two process agents resulting from the initialization phase to the execution phase.

By means of an appropriate labeling of control flow graph nodes in the graphical representation of Process1 (see Figure 1), we can directly relate the execution of individual operations to the resulting rules in the machine model.

$$PROCESS = \{ \mathsf{Process1} \}$$

$$execmod(\mathsf{Process1}) = \mathsf{Process1\_Module}$$

$$LABEL = \{ l_0, \ldots, l_3 \}, \quad startlabel(\mathsf{Process1}) = l_0, \quad inputlabels(\mathsf{Process1}) = \{ l_1 \}$$

Process1_Module:

$\textsc{ExecProcess}$
$\equiv$ `if` $label(Self) = l_0$ `then`
  $\textsc{AssignValue}((x, Self), eval(5, Self))$
  $label(Self) := l_1$

`if` $label(Self) = l_1$ `then`
  $\textsc{ConsumeInputSignal}(\{(A, \langle\rangle, l_2)\}, \emptyset)$

`if` $label(Self) = l_2$ `then`
  $\textsc{AssignValue}((x, Self), eval(x + 1, Self))$
  $label(Self) := l_3$

`if` $label(Self) = l_3$ `then`
  $\textsc{OutputSignal}(A, \langle\, \rangle, Env, undef)$
  $label(Self) := l_1$

`if` $label(Self) \notin inputlabels(Self)$ `then`
  $\textsc{HandleSignals}$

4.5. Interaction with the Environment

In Table 1, we summarize some interfaces to the external world. Signals, for instance, may either be introduced by the system or by the environment extending the domain *SIGNALINST* accordingly. Notice also that by declaring *SIGNALINST* as shared, functions defined on *SIGNALINST* are shared as well.

Regarding time related aspects, one can reasonably assume that values of *now* increase monotonically. Similarly, one can characterize all other interfaces, stating the expected behaviour of the external world (where the degree of detail and precision clearly depends on the underlying abstraction level).

Identifier	Type	Intended Meaning
SIGNALINST	shared	dynamically created signal instances
PID	shared	dynamically created PId's
now	monitored	global system time
delay	monitored	delaying behaviour of channels
schedule	shared	signal queues of (system) gates
Spontaneous	monitored	indicates spontaneous transitions

Table 1
Classification of dynamic ASM functions/predicates

5. CONCLUSIONS

In this paper, we have applied ASMs to define the dynamic semantics of a sample SDL specification formally. ASMs meet the design objectives of the dynamic part of the formal SDL semantics and have therefore been chosen as the underlying formalism. In addition, the static semantics as well as the data model have to be defined and combined with the behaviour model.

A particularity of the dynamic semantics is the compiler view, which — in conjunction with ASMs — yields an extremely concise behaviour model and will enable SDL-to-ASM compilers. Once such a compiler is available, existing ASM-tools can be used to execute the semantic model of any given SDL specification.

Readability of the semantic model is supported by a structure that mimics SDL structuring mechanisms. As a novelty, we have integrated the structured modeling of SDL communication links. This further improves readability and makes the semantics even more flexible w.r.t. future language development.

Two important observations concerning our modeling approach are in place:

- The SDL view of distributed systems with real-time constraints and the semantic modeling concept of multi-agent real-time ASM clearly coincide; essential properties of the underlying computation models — namely, the notions of concurrency, reactivity and time as well as the notion of states — are so tightly related that the common understanding of SDL can directly be converted into a formal semantic model avoiding any formalization overhead.

- Even without direct support of object-oriented features in ASMs, the resulting ASM model of the dynamic semantics is particularly concise, readable and understandable. Furthermore, this model can easily be extended and modified as required for an evolving technical standard. Beyond the purpose addressed here, ASM models can be utilized for defining a bridging semantics in order to combine SDL with other domain specific modeling languages.

ACKNOWLEGEMENTS

We take this as an opportunity to thank Joachim Fischer and his group for actively supporting our endeavour to define a new formal SDL semantics. We also thank Anders Olsen, Rick Reed, and Thomas Weigert for their advise and encouragement.

REFERENCES

1. E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998.
2. E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*. Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York (1999)
3. G. Del Castillo. The ASM Workbench: an Open and Extensible Environment for Abstract State Machines. In U. Glässer and P. Schmitt, editors, *Proc. of the 5th International ASM Workshop*, TR, pages 139–155, Magdeburg University, 1998.
4. U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 2, pages 271–280, 1998.
5. U. Glässer and R. Karges. Abstract State Machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.
6. U. Glässer, R. Gotzhein and A. Prinz. SDL Formal Semantics Definition. Technical Report tr-rsfb-99-065, Department of Mathematics and Computing Science, Paderborn University, 1999.
7. R. Gotzhein, B. Geppert, F. Rößler, and P. Schaible. Towards a new formal SDL semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 55–64, 1998.
8. Y. Gurevich. Evolving Algebra 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
9. Y. Gurevich. ASM Guide 97. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan–Ann Arbor, 1997.
10. Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In *Proceedings of CSL'95 (Computer Science Logic)*, volume 1092 of *LNCS*, pages 266–290. Springer, 1996.
11. St. Lau and A. Prinz. BSDL: The Language – Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995.
12. ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union (ITU), Geneva, 1994 + Addendum 1996.